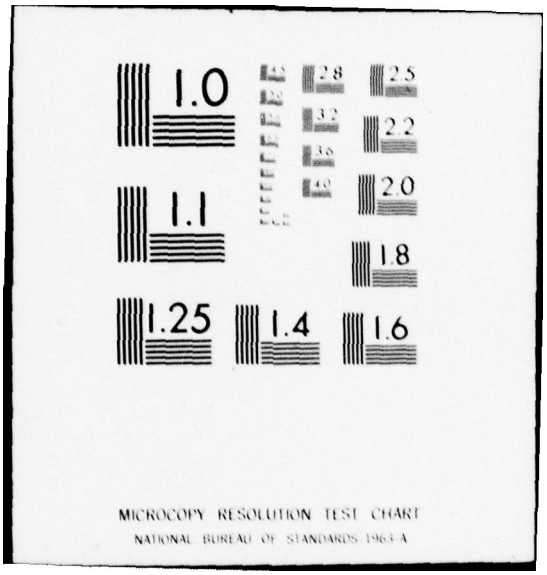
AD-A055 383 COMPUTER SCIENCES CORP HUNTSVILLE ALA F/G 9/2
HOL CODE GENERATOR DEVELOPMENT METHODOLOGY.(U)
NOV 77 J W CRENSHAW, D R GRIFFIN DAAK40-77-C-0048
UNCLASSIFIED CSC/TR-77/5496 NL

1 OF 1
AD
A055 383

END
DATE
FILMED
8-78
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

(1)

HOL CODE GENERATOR DEVELOPMENT METHODOLOGY

CONTRACT DAAK40-77-C-0048

Developed for

U.S. ARMY MISSILE COMMAND
GUIDANCE AND CONTROL DIRECTORATE
Missile Computer Software & Hardware Center
Redstone Arsenal, Alabama 35809

CSC/TR-77/5496

8 November 1977

CSC
COMPUTER SCIENCES CORPORATION

CSC

(1)

(6) HOL CODE GENERATOR DEVELOPMENT METHODOLOGY.

(9) Final rept.,

(15) CONTRACT DAAK40-77-C-0048

(12) 70 p.

Developed for

U.S. ARMY MISSILE COMMAND
GUIDANCE AND CONTROL DIRECTORATE
Missile Computer Software & Hardware Center
Redstone Arsenal, Alabama 35809

(14) CSC/TR-77/5496

(11) 8 November 1977

(10) J. W. CRENSHAW/D. R. GRIFFIN
COMPUTER SCIENCES CORPORATION
515 Sparkman Drive, NW
Huntsville, Alabama 35806

Major Offices and Facilities Throughout the World

D. E. Copeland
Technical Monitor
Missile Computer Software and Hardware Center
Guidance and Control Directorate
U.S. Army Missile Command
Redstone Arsenal, Alabama

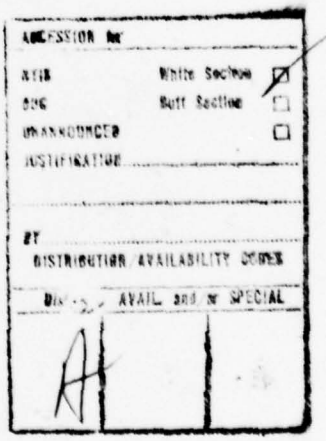


TABLE OF CONTENTS

LIST OF FIGURES

Figure

PREFACE

This final report is the result of a two man-month study to define a methodology for structuring the development of High Order Language (HOL) code generators for special Missile Systems applications. The report includes an analysis of code generation techniques in general, the special requirements of imbedded Missile Systems applications, and recommendation of an approach to providing effective compilers compatible with low-cost retargeting.

SECTION 1 - INTRODUCTION

1.1 BACKGROUND

In recent years there have been two significant trends in the DOD community which tend to alter the requirements for Higher-Order Language (HOL) compilers for digital computers: (1) The increasing use of "imbedded" digital computers in avionics, test hardware, and other devices, and (2) The recognition of the value of HOL programming, as opposed to a machine-oriented language (MOL), with respect to overall life-cycle costs.

Until fairly recently, the uses of HOL's were essentially limited to scientific research using large-scale general-purpose computers. For such uses, a compiler was both resident in and targeted (i.e. intended to generate code) for the general-purpose computer. For such an application, a considerable number of man-hours of labor could be expended in the development of a single compiler, since they were not written that frequently. Since the compilers were used in a batch mode, compiler size and/or efficiency were sometimes given heavy emphasis.

With the widespread use of small, special purpose flight computers in imbedded applications, the requirements have changed.

Imbedded computers tend to be too small and specialized to support resident HOL compilers. Therefore, the trend has been toward the use of "cross"-compilers resident on a large, general-purpose computer but targeted to the flight computer.

Since imbedded computer applications are almost always real-time and often time-critical, special emphasis is given to the generation of highly efficient object code. In this case, compiler efficiency is of lesser importance.

Each digital flight computer or other imbedded computer tends to be highly specialized with distinctive internal architecture and a unique instruction set. Because of technological advances, obsolescence is rapid and a given computer may only be used in a single program. In such a situation, it is wholly impractical to develop a complete HOL compiler for each target machine, even if the host computer is unchanged. This problem has been compounded by the advent of microprocessors and the recent proliferation of HOL's oriented for missile applications.

A partial solution to this problem is to recognize that many of the tasks performed by a compiler are done at the source syntax level, and are independent of both the host and target computers (and even the HOL being compiled). It is only the portion of the compiler which addresses the generation of actual machine code that need be altered. This suggests that it is possible to structure a compiler in such a way that this portion, the code generator, is modular and easily altered to adapt to a different target computer.

Unfortunately, although the code generator (CG) section of a compiler is generally a fraction of the total code and requires a much smaller fraction of the total execution time, it tends to be the least structured and orderly portion. In this respect it reflects the structure of the target machine to which it must ultimately conform. Also, optimization of the object code requires involvement of the CG in a machine-dependent way.

1.2 STUDY OBJECTIVES

This study represents a preliminary analysis aimed at the definition of a methodology for structuring the development of HOL code generators for missile systems applications. The purpose of this methodology is to provide a structured technique for providing low-cost, easily-retargetable compilers to support the software development for small missile applications.

Since the missile systems applications of greatest interest involve real-time operation, special emphasis will be placed upon the generation of efficient optimized code, particularly that for arithmetic expressions. The intent is to define techniques whereby effective optimization can be performed without requiring a high degree of target dependence in the code generator.

The following ground rules were adopted:

- The techniques proposed are to be applicable to existing compilers; it must not be necessary to develop a new compiler in order to realize some of the advantages.
- A non-optimizing compiler is not acceptable.
- Factors to be optimized are:
  - Retargeting cost
  - Execution time of object code, with emphasis on missile system applications
  - Program life cycle cost (impacted by both size and time efficiency)
- Factors which may be sacrificed are:
  - Compiler execution time
  - Compiler development cost

It should be noted that the factors to be optimized are, to some extent, mutually exclusive. For example, it is reasonable to suppose that a compiler designed for lower retargeting cost will require a different organization of the optimization algorithms, and thus will generate somewhat less efficient code.

1.3 SCOPE

The scope of this study is to define a methodology for structuring the development of High Order Language (HOL) code generators for special missile systems applications. This is accomplished by an initial analysis of the compilation process, followed by a discussion of conceptual enhancements to the process for retargeting which are relatively obvious from the analysis.

The process of HOL compilation is analysed in Section 2. The flow from HOL Code to machine code is discussed including syntax analysis, use of intermediate languages, optimization, and code generation.

In Section 3, it is pointed out that for the use of HOL's in missile system applications, it is not necessary to redevelop the HOL compiler for each new missile system. The use of assembly language outputs and multiple intermediate languages with regard to the problems of code generators and retargeting is a viable and economic alternative.

There are current tools available for both retargeting and rehosting. The current practices and some of the tools are discussed in Section 4.

These discussions lead up to a set of conclusions and recommendations pursuant to a methodology for code generators for missile system applications which are presented in Section 5.

SECTION 2 - ANALYSIS

2.1 COMPILATION

The process of compilation is one in which an HOL source code (consisting of statements written in the syntax of the HOL) is converted into a machine-oriented object code. In essence, this process is one of string replacement: the character string representing the source code is subjected to a series of transformations, according to a set of replacement rules, until it becomes the output string representing the object code.

In general the direct conversion of a statement or series of statements directly to object code is much too formidable to attempt in a single step. Instead, the problem is made amenable to solution by breaking it down into several smaller steps. A typical sequence is symbolized by Fig. 2-1. In the Syntax Analyzer (parser) section each source language statement is scanned, the operators and identifiers are isolated, and the statement type is identified. Incorrectly formed statements are detected here. The source statements are reformatted to save space and tagged with various indicators giving, for example, the statement type. During the process of parsing, variable and subroutine names are identified and added to a symbol table. In some compilers, the statements are reordered to some extent so that all statements of a given type can be compiled together. Parsing is probably the most time-consuming part of the compilation process. However, it can be made quite orderly. In fact, automated compiler-compilers exist which can generate a parser from the syntax definitions of a language.

If global optimization is to be performed, it is done at this level. Typical of this optimization is the reorganization of the program flow to remove unused or repeated code, and to eliminate common subexpressions.

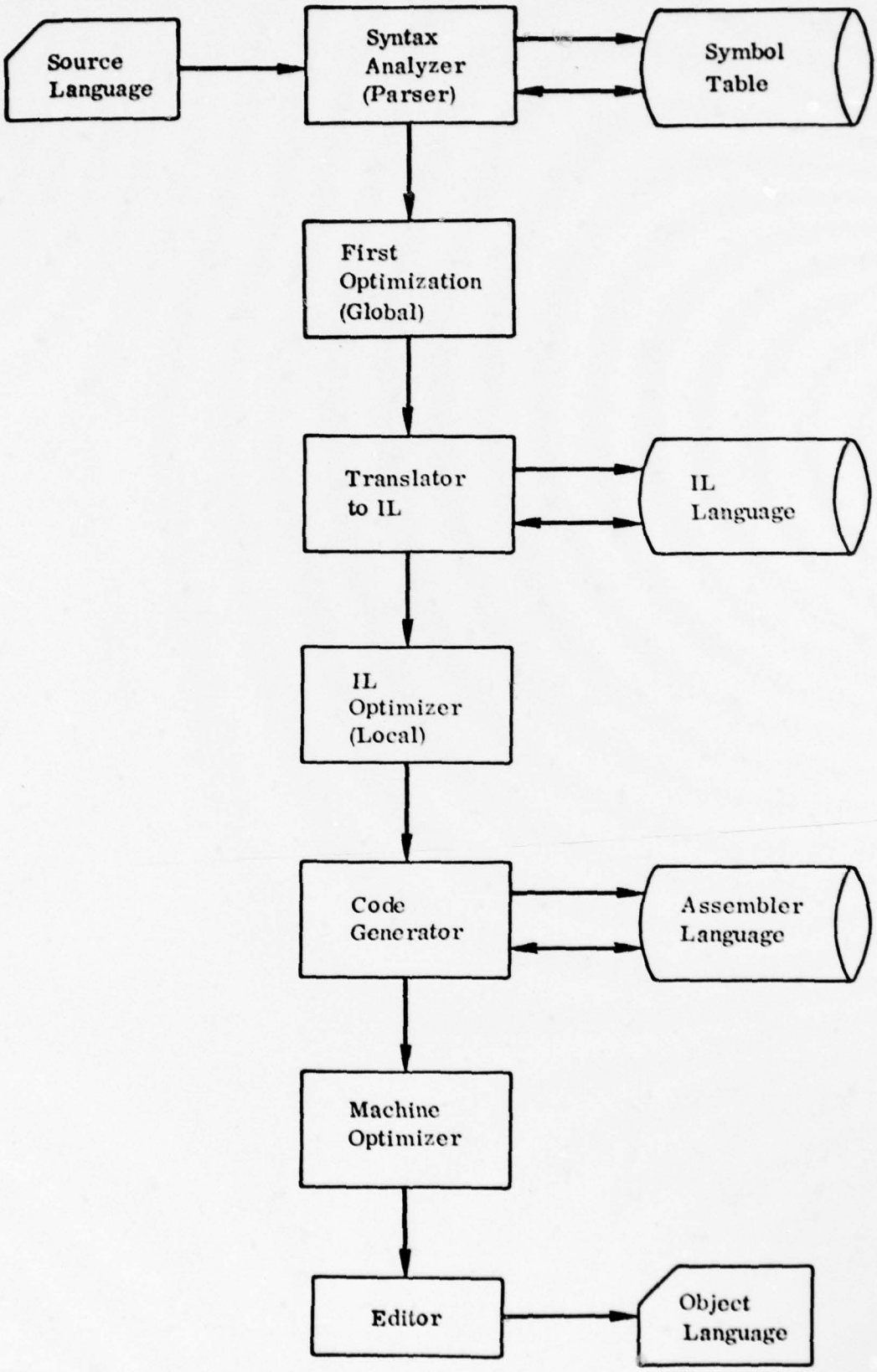


Figure 2-1 The Compilation Process

The structure of the source language is one that is intended to be convenient for the user, but not necessarily for the computer. In particular, a single source statement generally calls for many distinct operations to be performed. Before generating object code, it is first necessary to identify these operations and their order. To do this, it is common to define an Intermediate Language (IL) in which each separate operation corresponds to a single IL instruction. Note that a single IL operation may (and usually does) require more than one machine instruction. For example, a floating point multiply may be represented by a single IL instruction, even though it cannot be directly performed by hardware and requires a subroutine call to realize. The IL "operation" is chosen to be any sequence of instructions that can conveniently be isolated as a self-contained, independent sequence. The language chosen for the IL is still typically quite abstract. Symbolic addressing is used and little or no consideration is given to actual machine characteristics such as word length, number of registers, etc.

Following translation to the IL, there are several further optimizations that can be performed at the IL level. For the most part these are local optimizations.

The next step in compilation is to replace each IL statement by an equivalent sequence of machine-level instructions. This step is performed by the code generator. Normally, in this step symbolic addressing is still retained. In a sense, then, the code generator outputs assembly language for the target machine. Indeed, in many compilers, particularly early ones, this was the last step performed. The assembly language was then passed to a separate assembler. In most cases, this is no longer done. The reason is that a separate assembler must repeat many steps, such as building a symbol table, that have already been performed by the parser. In a compiler, the only "assembly" task to be performed is that of assigning specific locations to the symbolic parameters. This task is performed by an Editor.

Between the CG and the Editor, a final phase of global optimization is performed. The most signification optimization done here is that of register management, assigning parameters to registers in such a way that repeated access to often-used parameters can be done efficiently.

The process of compilation can be thought of as a transformation from a higher order of abstraction (HOL) to a low order. This is symbolized on an "abstraction continuum" in Fig. 2-2. As indicated, each segment of the compiler serves to reduce the level of abstraction, which is finally reduced to absolute machine code by the Relocatable Linking Loader.

The portion of the compiler that is of concern for retargeting is any portion which is machine-dependent. In Fig. 2-2, this is primarily the CG (shaded in the figure) but actually can include everything to the left of the IL level. One observation may be made here. It appears that one reasonable approach to reducing the cost of the code generator is to reduce its responsibility by moving functions either into the Translator or the Editor/Assembler.

2.2 INTERMEDIATE LANGUAGE

The choice of an intermediate language for a compiler is largely arbitrary, and many such languages have been used. However, it should not be surprising that the choice of intermediate language can have a profound effect on the compilation efficiency, the execution efficiency and the organization of the compiler itself. In early compilers, a common IL was Polish (Lukasiewicz) notation. This was primarily because Polish notation is easily derived from ordinary algebraic notation. Each operator in Polish operates on only one or two parameters, and so the notation meets most of the requirements for an IL. Unfortunately, Polish notation is basically a stack-oriented language, whereas all real computers are essentially register-organized (even though they may support certain stack operations). Therefore, a considerable amount of translation is required in the code generator to convert Polish into machine code.

Figure 2-2 Levels of Abstraction

Other forms of IL are the doublet, triplet, and quadruple forms. In these IL's, each IL instruction consists of an instruction operation code, followed by one, two, or three operands. A typical sequence of doublets may be:

```
LOD     X
ADD     Y
STO     Z
MULT    P
SUB     Z
OUT
```

(In reality, of course, numeric operation codes would be substituted for the mnemonics shown.) A single accumulator is assumed so that each operation is assumed to be upon the accumulator and the named operand. Note that this language may, in fact, be very nearly equivalent to assembly language.

Both Polish notation and the doublet form are best suited to single-accumulator machine, in which all operations must be performed on the data in this accumulator and the result left there. Most modern computers, however, contain more than one general-purpose register, at least some of which function as accumulators. For this reason, most compilers now use the triplet or quadruple form. In the quadruple form, the three operands represent the two parameters operated upon, and the source parameters to which the result is assigned. In the triplet form, the source parameter is omitted; the result is assumed to be left in the location assigned to one of the input operands.

In selecting the form for the IL, the architecture of the target machine should be taken into account. Thus, the doublet form may be very efficient for a single-accumulator computer, but very poorly suited for one with memory arithmetic (that is, each memory location functions as an accumulator). For the latter, a triplet form would be preferred.

Tailoring the IL to the target machine is both possible and desirable for a resident compiler. It is neither for a retargetable cross-compiler. This places even more emphasis upon choosing an IL that is relatively general. It also suggests greater inefficiency, at least in compile time, and probably in object execution time as well.

Having selected an instruction format for the IL, there is still the question of the IL instruction set. This is a matter of judgement and experience. There is no optimum set, since the ease of translation depends upon the particular problem being solved. However, the IL instruction set should offer a reasonable correspondence with both the operations available in the source language and the target instruction set. Once again, for a multiple-target situation, greater compromise is likely.

2.3 CODE GENERATION

Conceptually, the process of code generation is very straightforward, and can be carried out by a method of simple string substitution. For example, the IL quadruple

ADD A, B, C

may be replaced by the assembly language instructions

```
LOD  A
ADD  B
STO  C
```

The code generator must simply replace one string by another, and substitute the operand identifiers in the proper places, as indicated by the underlines.

A simple, "quick-and-dirty" code generator can, in fact, be written in just such a fashion. Since it can be written as a table-driven algorithm, retargeting would be simply a matter of replacing the tabular data. This approach is actually taken when rapid development of a new compiler is required. Since the CG is table-driven, it is relatively easy to devise automated methods for developing the CG itself.

The difficulty arises when we also require efficient object code. Consider, for example, the FORTRAN expression

$$X = (A + B)/C \quad ,$$

which would translate to the IL sequence

```
ADD A, B, TEMP
DIV TEMP, C, X .
```

Each quadruple may be replaced by a set of three assembly language instructions

```
      LOD     op  1
operation     op  2
      STO     op  3
```

(Here a typical set of mnemonics for a single-accumulator machine is assumed.) Direct replacement of the two quadruples would then yield:

```
LOD   A
ADD   B
STO   TEMP
LOD   TEMP
DIV   C
STO   X
```

The two instructions indicated are superfluous, and would be omitted by a human programmer. Thus a straightforward code substitution tends to lead to very inefficient code.

It has often been said that a good assembly language programmer can write more efficient object code than the best compiler, and this is in general quite true. There are a number of subtle ways in which such a programmer can utilize coding "tricks" to improve the efficiency of his code. For example, the optimum method of implementing a given source statement may be quite different, depending upon the given problem and what takes place in neighboring statements. Such subtlety is impossible for a simple, substitution-oriented CG. It is the attempt by the compiler developer to inject some of this subtlety into the code generation process that results in a complicated, highly machine-dependent CG.

2.4 OPTIMIZATION

From the discussion so far, it appears that the degree of optimization and its implementation has a profound effect upon the practicality of achieving the simultaneous goals of low-cost retargeting and efficient object code. Therefore the techniques available for optimization are examined next.

2.4.1 Local Optimization

To illustrate some of the techniques for local optimization, let us consider the FORTRAN instruction

$$X = (A + B)/C - (D * E + F * G)$$

Straightforward translation of this statement yields the IL sequence

```
ADD     A, B, TEMP1
DIV     TEMP1, C, TEMP2
MUL     D, E, TEMP3
MUL     F, G. TEMP4
ADD     TEMP3, TEMP4, TEMP5
SUB     TEMP2, TEMP5, X
```

Converting to object code as before would give

```
LOD     A
ADD     B
STO     TEMP1
LOD     TEMP1
DIV     C
STO     TEMP2
LOD     D
MUL     E
STO     TEMP3
LOD     F
MUL     G
STO     TEMP4
LOD     TEMP3
ADD     TEMP4
STO     TEMP5
LOD     TEMP2
SUB     TEMP5
STO     X
```

This is a sequence 18 instructions long. The code can be shortened by noting that the indicated STO/LOD pair is superfluous. It is relatively easy to design the CG (or Editor) to scan for such unnecessary pairs and delete them. Although there are other such pairs, they have different operands and cannot be deleted. However if, following the deletion of the first pair, we exchange the operands of all commutative operations (ADD and MUL) we have

```
LOD     A
ADD     B
DIV     C
STO     TEMP2
LOD     E
MUL     D
STO     TEMP3
LOD     G
MUL     F
STO     TEMP4
LOD     TEMP4
ADD     TEMP3
STO     TEMP5
LOD     TEMP2
SUB     TEMP5
STO     X
```

Now the TEMP4 pair (and the intermediate variable as well) can be deleted. Finally, by permitting the unary minus operation (the mnemonic CHS is assumed), the subtract can also be reversed. The resulting code is

```
LOD     A
ADD     B
DIV     C
STO     TEMP2
LOD     E
MUL     D
STO     TEMP3
LOD     G
MUL     F
ADD     TEMP3
SUB     TEMP2
CHS
STO     X
```

This is a sequence of 13 steps, a reduction of 28% over the original. An experienced assembly language programmer would probably have written

```
LOD   D
MUL   E
STO   TEMP1
LOD   F
MUL   G
ADD   TEMP1
STO   TEMP1
LOD   A
ADD   B
DIV   C
SUB   TEMP1
STO   X
```

which is only one instruction (8%) shorter than the "machine coded" version. This result suggests (but certainly does not prove) a principle that seems to hold for HOL's. If local optimization is applied, mathematical assignment statements tend to compile very efficiently, and will run little, if any, slower than the same expression coded by hand. Since the great majority of HOL code for missile system applications is mathematical, there is reason for a certain optimism that the goal of efficient object code can be met without complex optimization schemes.

It should be noted that the process is not as simple as has been implied. For example, if the term A + B is used elsewhere, then it would be incorrect to delete the STO TEMP1 in the first pass. The LOD TEMP1, however, can be deleted. It should further be noted that some of the steps outlined here can be performed at the IL level. Thus an optimizing translator would have output the IL sequence

```
ADD   A, B, *
DIV   *, C, TEMP2
MUL   D, E, TEMP3
MUL   G, G, *
ADD   *, TEMP3, *
SUB   TEMP2, *, X
```

where the asterisk (*) represents the accumulator. The CG is still required to recognize the * as an indication to skip the corresponding LOD or STO, and, in the last instruction, to insert a CHS.

Two other forms of local optimization are strength reduction and reduction of common subexpressions. The former technique involves the identification of constant subexpressions and evaluation of them at compile time. Thus

$$X = (2 + 3) - 4$$

would be compiled to

```
LOD  1
STO  X .
```

Note that this requires the compiler to evaluate, as well as compile, arithmetic expressions. In the case above, the result is obvious. In other cases, particularly those involving loop indices, it is much more subtle. Strength reduction is most effective in this area and, since missile system applications tend to require loops and indexed variables, its use is highly recommended.

The second technique involves scanning the source code for common expressions. For example, consider

$$X = (A+B)/(A-B) - (A-B)/(A+B) .$$

Translation to IL would yield

```
ADD   A, B, TEMP1
SUB   A, B, TEMP2
DIV   TEMP1, TEMP2, TEMP3
SUB   A, B, TEMP4
ADD   A, B. TEMP5
DIV   TEMP4, TEMP5, TEMP6
SUB   TEMP3, TEMP6, X
```

By scanning the operator and first two operands, we find the two operations ADD A, B and SUB A, B are performed twice. The second of each can be deleted by replacing its third operand. Thus

```
ADD     A, B. TEMP1
SUB     A, B, TEMP2
DIV     TEMP1, TEMP2, TEMP3
DIV     TEMP2, TEMP1, TEMP6
SUB     TEMP3, TEMP6, X
```

Note that this process can be extended globally by extending the region of scan. However, in this case the compiler must verify that A or B (in this case) are not altered between appearances of the subexpression. Thus the compiler must keep a record of the update status of each variable. Two subexpressions are common only if none of the variables within them have been altered.

In the case of loops and transfers, the status of a variable at some point may depend upon the path taken. For this reason, common subexpressions are generally scanned only within blocks of in-line code. In some cases, this reduces the potential gains from the optimization.

Note that both strength reduction and removal of common subexpressions are performed at the IL syntax level, and need not impact the development of a code generator.

2.4.2 Global Optimization

There are many forms of optimization that must be performed globally ... that is, over more than one source statement. One example, common subexpression removal, has already been discussed. Another is the removal of constant expressions from loops. Note that this is not the same thing as strength reduction. An expression may involve parameters which are globally variable. However, if these variables are not altered within a loop, the expressions can be moved outside the loop. Note also that this process involves rearranging the source code, rather than simply converting it to efficient object code. This rearrangement is characteristic of global optimization.

It should be noted that the optimization techniques mentioned so far, except the STO/LOD suppression, can also be performed manually, by the programmer. This gives rise to an observation:

Prior to the development of optimizing compilers, the writing of efficient HOL was left to the programmer, and certain technique were taught to achieve this. These included avoidance of common subexpressions by the use of intermediate variables, precomputation of constant terms, removal of constant expressions from loops, etc. Such techniques were then considered good programming practice. In that sense, the global optimization techniques mentioned serve to correct the "errors" of the programmer.

On the other hand, the current trend is to encourage the programmer to write code in a straightforward manner, without use of any "tricks" to achieve efficiency. This is now left to the compiler. Thus the definition of "good programming practice" has been changed. There can be little doubt that this is in the long-term best interest of the state of software development. In keeping with the trend away from machine-level languages and toward abstract ones, the programmer should not be required to use certain constructs just to satisfy efficiency constraints. A program written in a straightforward manner is more easily maintained than one written with coding tricks.

However, there is certainly a limit as to what can be expected of an optimizer. It cannot be expected to be 100% efficient. Therefore it seems axiomatic that an HOL program written to be efficient will be more so than one written in sloppy fashion and then machine-optimized.

In the final analysis, the choice, as usual, involves a tradeoff. What is desired as a result of this study is a compiler which can be easily and quickly retargeted, which can generate highly efficient object code, and which does not require efficiency of coding by the programmer. These goals are to some extent mutually exclusive. In any such tradeoff, the cost of retargeting will be given more weight here.

One final observation can be made. With the one exception of register management, the global optimization techniques can be performed at the source level and will not impact the development of the code generator. Since compile-time efficiency is not an issue here, and since global optimizations can have significant impact upon object execution time, it is advisable to make the maximum use of source-level, global optimization. Some enhancements to normal levels of optimization are discussed in Appendix B.

2.4.3 Register Management

The exception mentioned above, the global optimization which is machine-dependent, is that of register management. It is also one that cannot be omitted, since it is one that can have profound effects upon execution time.

In single-accumulator machines such as were common in the early days of computer technology, there was no management problem, since there was no choice as to register storage. This made the compiler's task fairly straightforward. Current missile flight computers typically have several registers, more than one of which may be a general-purpose accumulator. In addition, there may also be a high-speed cache memory for frequently-used data.

The assignment of global or temporary variables to the various accumulators, registers, index register and memory tends to be highly machine dependent. Often certain operations can be performed in certain ways, while others cannot. For example, one microprocessor permits direct increment or decrement of memory, but not add to memory. It also permits a compare of memory to the accumulator, while others require the item of memory to be loaded into a register first.

Overlooking the machine ideosyncracies for a moment, consider the problem of management of storage itself. The first question to be asked regarding any variable is whether it should be stored at all. This touches on the local removal of STO/LOD pairs mentioned earlier.

A variable need only be stored if it is to be used later than the next operation, or if it needs to be in a different register for the next one. The process of determining variable usage is called a Dead Variable Analysis. It has been typically used for compressing data storage in main memory, but the same approach applies to register storage as well. It involves the building of a "Set/Used Table" which gives the status of each variable. After the last usage of a variable, or the last usage before it is reset, it is dead, and another variable may be stored in its place.

Once the storage history of the variables has been determined, they must be assigned to storage. For this purpose, the frequency of use must be examined. Variables which are used often should be stored in register memory, while those used least often should be in main memory. The type of usage should also be considered For example loop counters are obvious candidates for storage in index registers. In the frequency of usage analysis, the program should be considered in blocks. It may be useful to provide both register and main memory storage for a variable. For example, it may have a high frequency of usage within a loop. In such a case it would be convenient to move the variable to a register before beginning the loop, and move it back to main memory upon loop exit. Obviously, the compiler must provide for proper update of memory for all possible exit paths from the loop.

While, as mentioned, the optimum assignment of storage can be highly machine dependent, it appears that the dead variable analysis can at least be partially mechanized in a regular, parameterized fashion, i.e. for a machine with n accumulators, m registers, k index registers, etc. There will probably remain a certain amount of register assignment that must be handled differently for each target machine. One (non-optimal) way to handle this is to assign specific registers for certain special tasks (e.g. index 4 for subroutine offsets, index 1 for loops).

SECTION 3 - ENHANCEMENTS FOR RETARGETING

In the situation of interest in this study, it is assumed that it is not necessary to develop a complete new compiler. Rather, it is assumed that an HOL compiler exists and is resident on the host machine, and it can be modified to meet the goal of low-cost retargeting. Some modifications and enhancements capable of reaching this goal are discussed in this section.

3.1 ASSEMBLER LANGUAGE OUTPUT

Recall that in Fig. 2-2, all portions of the compiler to the left of the IL level are (or may be) target machine-dependent. If the code generator outputs to an editor rather than an assembler, this editor is target-dependent and the cost of retargeting the editor must be considered along with that of the CG. A similar statement holds for the linking loader.

If retargeting of the editor and loader can be removed from consideration, the cost of retargeting can obviously be cut. This is possible if a stand-alone cross-assembler and loader can be assumed to be available for each potential target machine. This is a reasonable assumption. All manufacturers of commercial computers and micro-processors offer cross-assemblers with macro capabilities. Most are written in FORTRAN IV for portability. There is a charge for these assemblers of about $500 - $2500, but this is small compared to the cost of retargeting the editor. The only case in which a cross-assembler is not likely to be available is for a bit-slice, microprogrammed machine with a custom instruction set, for which no assembler has been written. This is an unlikely occurence. In any case, if a computer with a non-standard instruction set is selected, the cost of developing an assembler should really be charged to the hardware development effort, rather than to software.

One consideration should be borne in mind. Although cross-assemblers are generally available for the computers of interest, they are not uniformly powerful. They tend to have rather limited macro facilities and limited diagnostics. Since, however, the input to the assemblers will be machine-generated, this is not seen as a problem.

3.2 MULTIPLE IL'S

As mentioned in Section 2.2, the typical compiler IL is quite abstract. Although it does have one instruction for each distinct operation, the instructions and format do not correspond to any potential target machine. The addressing is symbolic, and no regard is given to actual machine considerations such as word length, storage format, register available, etc. To do so would inject machine dependencies into the translator section of the compiler, which is clearly undesirable. However, the degree of abstraction associated with typical IL's places a greater load on the task of code generation than is necessary.

Referring to Fig. 2-2, suppose that the code generator could be split into two parts, one of which is machine-dependent, the other of which is not. Then only the second portion need be involved in retargeting. One way of doing this is to define a second IL, less abstract than the first, and one more closely akin to machine language. The situation is symbolized in Fig. 3-1.

In effect, this approach is equivalent to defining a fictitious computer, an "abstract machine" which, if mechanized, would directly execute the instructions of the second IL. This concept of an abstract machine is well established in the literature (1), (2), (3), (4). The ordinary IL of any compiler may be regarded as the two-or three-address machine language for an abstract machine. In fact, any language, including the original HOL, may be associated with an abstract machine. Depending upon the method of implementation, the tool for conversion from the abstract machine language to the target language may be referred to as an emulator, interpreter or, as in the current case, a code generator. The concept of multiple IL's has recently received considerable attention in the development of interpretive languages for microprocessors (6), (7), (8).

For the purposes of the current study, it is suggested that the second IL take the form of a "Universal Assembler", language whose instructions are at the machine operation

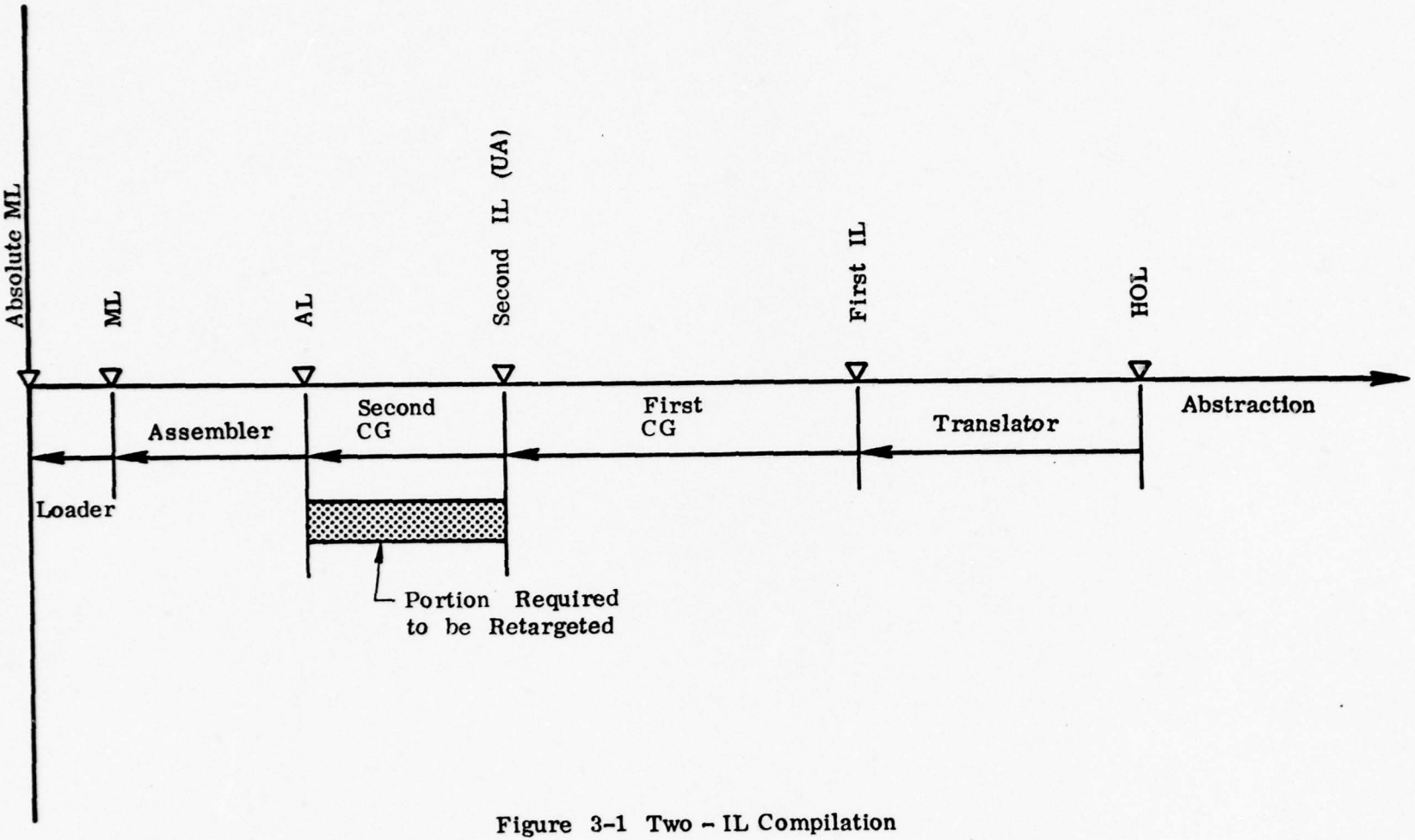


Figure 3-1 Two - IL Compilation

level, but do not necessarily correspond to any actual machine. Such a language, called "MIMIC", is actually in use in CSC software development activities. For the selection of the universal assembler (UA) language it is necessary to define an "Ideal Machine", with an instruction set that represents a rational set of machine-level instructions, leading to efficient "object" code (note that all real computers should be, but rarely are, designed in this manner). The code generator task then becomes a simple macro substitution of target machine instructions for the UA "pseudo-ops".

For this approach to result in reasonably efficient object code, the instruction set for the UA must be carefully selected. For example, the instruction set for the PDP-11, like many IL's, is basically two-address. If the UA were chosen to correspond to a single-address ideal machine, the resulting code may tend to be inefficient. It is difficult to imagine a single UA which provides a good match to all computers. Fortunately, it is only necessary to generate efficient object code for a special subset: those computers which are candidates for missile flight systems. It is suggested that the instruction sets of these candidates be examined in order to synthesize the UA instructions. Within DOD an effort is underway to define a standard instruction set for all future DOD applications. The instruction set for this "Software Compatible Family" is an obvious choice for the UA.

Although the UA, and the first code generator which translates to it, is intended to be machine-independent, there are some cases in which this should not be strictly enforced. For example, if the target machine is a 16-bit machine, then the UA should also be, to avoid extensive data translation requirements. It appears that such considerations can be treated parametrically, by storing the machine-dependent parameters in an easily altered table. Machine dependence of this parametric type need not be feared, and is consistent with a parametric approach to optimization.

SECTION 4 - CURRENT PRACTICE

An example of the current practice in compiler development may serve to suggest some parallel approaches for the current needs. Such an example is provided by CSC's development of the J-3B and J-73 compilers for the Air Force. These compilers were developed specifically for dedicated avionics applications, and feature:

- o High level of optimization
- o Heavy use of compiler development tools
- o Developed directly from source language syntax specifications
- o Well-defined IL and CG interfaces
- o Designed for efficient rehosting and retargeting.

Some tools associated with these compilers and their development are discussed next.

4.1 SYMPL

Systems Programming Language (SYMPL) is a CSC - proprietary programming language developed using systems programming aspects of FORTRAN and PL/1. It has been in use for over 13 years in the development of compilers. The production versions of CSC - developed compilers are often written in SYMPL, which are then compiled into object code.

4.2 GENESIS

This CSC - proprietary language is a compiler development tool, useful for compilers which can be written in a table-oriented form. Based upon the defining syntax specifications for the language, GENESIS generates the tables and connecting software which serve to implement the syntax analyzer section of the compiler. The output of GENESIS is SYMPL source code.

4.3 JOCIT

For the automated development of J-73, the JOCIT program was developed. This program is both a JOVIAL compiler and a compiler-compiler. A complete description of JOCIT and the requirements for retargeting it are given in Appendix A. Some optimization enhancements to JOCIT are discussed in Appendix B.

4.4 ACG

The Automated Code Generator (ACG) is a CSC-proprietary program developed to achieve rapid retargeting for JOVIAL compilers. The program is designed to aid in the development of quick-response code generators of a table-oriented nature. The resulting "Quick code" generators are typically used as interim generators until more efficient retargeting can be effected.

4.5 AUTOMATED TOOLS

As can be seen from the descriptions just given, the current state of the art involves the extensive use of automated software tools to develop other software. With the aid of these tools, it is perfectly feasible to fully develop a compiler from its syntax definition, without ever working in the assembler language of the host machine. This approach is clearly in keeping with the recognized advantages of HOL programming.

SECTION 5 - CONCLUSIONS AND RECOMMENDATIONS

On the basis of the analyses presented in this report, it appears that it is feasible to modify an existing HOL to provide low-cost retargeting, without severely degrading the object-time execution efficiency. Some specific recommendations have been identified and are summarized below.

5.1 COMPILER ORGANIZATION

The compiler chosen must have an architecture such that there is both a well-defined intermediate language (IL) and a modular code generator (CG). An IL composed of triplets or quadruples is preferred. That portion of the compiler not included in the CG or Editor must not be target dependent.

5.2 OPTIMIZATION

The compiler chosen must have extensive capability to perform global optimization. It is suggested that existing capabilities be enhanced to include code straightening, dead variable analysis, and loop optimizations. The important area of register management should be performed in a parametric manner, so that retargeting can be accomplished by loading a table of machine parameters.

5.3 ASSEMBLY LANGUAGE OUTPUT

The code generator should output a character string consisting of assembly language for the target machine. Assembly and linking shall be accomplished off-line by separate software.

5.4 MULTIPLE IL'S

At least one extra level of IL should be considered. This should consist of a universal assembler (UA) language whose instructions represent assembly language instructions for a fictitious, "ideal machine." In the selection of this language, it is suggested that

candidate missile system computers, particularly the "Software-Compatible Family", be examined. Machine dependencies such as word length and number of registers should be parametric.

5.5 AUTOMATED TOOLS

It is suggested that all automated software tools available such as compiler-compilers be used. The availability of such tools should be a factor in the choice of the base HOL compiler.

APPENDIX A

JOCIT

A.1 BACKGROUND

The original intent of the JOCIT effort was to develop a compiler building tool for the J73 language. When the J73 contract was awarded, the language still was not firmly defined, and after a few weeks of study of the language it was determined that the development costs were higher than originally estimated. Therefore, in order to produce a more useful product that demonstrated the principal elements of the original objectives, the project was redirected towards developing a J-3 compiler building tool. Since the need within the Government for a J-3 capability was sufficiently real and urgent, the new project goals were highly practical. The principal objectives of the JOCIT J-3 development were to:

- Reduce the time and cost of implementing and maintaining JOVIAL J-3 compilers
- Ensure that JOVIAL language sets implemented on different computers are consistent
- Enable the rapid inclusion of any new JOVIAL features into every compiler built with the tool, including those compilers implemented before the feature was accepted
- Enable the compilers built with the tool to incorporate modern optimization techniques that overcome many forms of poor programming

Although the redirection to produce a running, debugged, efficient, and reliable J-3 compiler necessarily had the effect of diluting some of the goals of the tool,

nevertheless, the objectives were largely met. The most objectively measurable result of the JOCIT effort was the development of a production-grade J-3 compiler currently in heavy operational use. However, it is somewhat difficult to assess the result of the tool development in equally objective terms, because no new retargeting or rehosting effort has been undertaken. The following paragraphs describe the essential features of the JOCIT program and assess the relative success in meeting the stated goals.

A.2 DESIGN FEATURES

The following design elements were incorporated into JOCIT:

- Target-machine-dependent code isolated into functional modules
- Global optimization techniques to meet the requirements of language independence, host-machine independence, and target-machine independence
- The GENESIS system used for writing the J-3 language specification, the resulting tabular form of which is processed by a language-independent analyzer program
- A prototype compiler to compile the full J-3 language; the prototype can be used as a model for rehosting the J-3 version or as a basis for building a J73 JOCIT
- Over 95 percent of the JOCIT code written in an HOL (SYMPL); use of machine code is restricted to host-machine-dependent interfaces

A.3 CHARACTERISTICS OF THE JOCIT J-3 COMPILER

JOCIT embodies the following three features which, together, realize the goal of a tool for the generation of standard JOVIAL J-3 compilers:

- JOCIT is a stable, well-debugged, efficient, production-quality J-3 compiler. It realizes the most advanced optimization in any

JOVIAL J-3 compiler to date, and even though the compiler is large (40-50 K words of HIS-6000 main memory), it is quite fast and generates extremely informative and useful listings.

- Retargeting of JOCIT is a known, relatively straightforward, but not trivial process. It is achieved through total replacement or partial modification of certain compiler modules, and the installation of the JOVIAL library on the new target machine.

- Rehostability of JOCIT is achieved principally by programming the JOCIT modules in SYMPL. Rehosting is considerably more complex than retargeting, but the steps are well understood and meet the tool requirement by costing only a fraction of a comparable, totally new compiler implementation.

These three considerations are addressed in the following subsections.

A.3.1 USER INTERFACE

The JOCIT J-3 compiler is operated in a standard fashion entirely compatible with other GCOS language processors. That is, the command syntax and file specifications conform to GCOS standards, and the JOCIT user is required to learn only the computer options in order to invoke the compiler.

A.3.2 THE J-3 LANGUAGE

JOCIT implements the full J-3 language, with certain extensions added to satisfy unique customer requirements (including a special source language I/O facility to satisfy a user requirement for compatibility with the nonstandard Honeywell J-3 compiler). The diagnostic capability is thorough, and extensive use is made of parameterized diagnostics (for example, providing for insertion of identifier or reserved word names).

A.3.3 COMPILER LISTINGS

The JOCIT compiler provides a comprehensive set of compiler listings. These listings include interspersed Phase I diagnostics; a consistent diagnostic format

for all phases; an extensive object program assembly language-format listing identical to GMAP, a complete "set-used" listing for all program constructs (define names, status constants, program variables, labels, procedures, etc.); and a program environment listing.

A.3.4 OBJECT PROGRAM EFFICIENCY

The JOCIT program expends much effort to obtain object program efficiency. This has been achieved through two means: global, target-machine-independent optimization, and a code generation scheme that optimizes register usage and performs considerable special case analyses. Global optimization includes:

- Elimination of redundant common expressions computations
- Redistribution of loop-constant code
- Reduction of formal loop operator strength
- Improvement of compile-time constant arithmetic and subexpressions (e.g., elimination of multiplications by 1)
- Recognition of dead code
- Evaluation of compile-time constant predicates (e.g., AA = 1$...IF AA$)

All computational memory is embodied in the optimizer-code generator file (IL) interface, while local optimizations are performed by the code generator to produce the optimum sequence for each recognizable case. The combination of global and local optimization, which attempts to minimize generated code space, is successfully realized in the JOCIT J-3 compiler. Further improvements that would have a high payoff in production programs are:

- Regional index register dedication
- Loop control variable (LCV) index register dedication
- Improved strength reduction, including test replacement and dead LCV elimination

- Dead variable analysis
- Code straightening

Of the 60 percent improvement in generated code over the previous HIS J-3 compiler, 10 to 15 percent is attributable to global optimization, while the balance is derived from the local code generation algorithms. Even though there is room for improvement, the level and quality of the realized optimization are the notable achievements of the JOCIT J-3 model.

A.3.5 COMPILER EFFICIENCY

Considering its optimizing capabilities, the JOCIT J-3 compiler is comparatively fast. However, the compiler's instantaneous main memory requirements are quite high; 40K words is the minimum partition plus the size required for the compiled program's symbol table. The compiler already is heavily segmented into separate overlay loads. Only a radical redesign could reduce the core requirements, and only at the cost of severely reduced compiler speeds. The large size of the compiler is due to the following reasons (listed in descending order of impact):

- Complexity - The JOCIT J-3 compiler was designed for maximum user utility. It performs an enormous number of complicated tasks in order to produce pinpoint diagnostics, to perform global flow analysis/optimization, to pack tables optimally, and to provide a sophisticated and useful COMPOOL facility. The augmented J-3 language it compiles is huge, and the code generator achieves its goals through complex algorithms that require a considerable number of source code lines to effect. It is doubtful that the number of lines of code in the compiler itself could be materially reduced without seriously compromising user convenience and object code performance.

- Compiler Architecture - The compiler uses a minimum number of phases and intermediate files and requires a large symbol table to be resident throughout the compilation process. It is conceivable that by partitioning the compiler into more functional phases (a multipass code generator is a possibility) the maximum phase size could be reduced; but, as pointed out earlier, compiler speed would be reduced and additional program complexity (more intermediate files, for example) would result.

- Use of HOL - Because of the JOCIT J-3 compiler is written in SYMPL and compiled by a small compiler which incorporates only a modest number of local optimizing algorithms, the JOCIT compiler contains more lines of object code than would be the case if it had been written in assembly code or compiled by a sophisticated, optimizing SYMPL compiler. A more compact compiler also could be achieved by rewriting the JOCIT compiler using J-3, thereby obtaining the benefits of the compiler's own optimization. However, the J-3 language is much less suited to compiler implementation than is SYMPL and carries excess baggage (e.g., fixed point arithmetic, lengthy prologues and epilogues) that is costly and unnecessary. Optimizations could be added to the SYMPL compiler to reduce object code without unduly compromising rehostability or retargetability of either SYMPL or JOVIAL.

A.3.6 DEBUGGING (USER)

The inclusion of the MONITOR statement and ENCODE/DECODE provide considerable debugging convenience for the user. In particular, the availability of the compiler command option to suppress compilation of all MONITOR statements allows the user the convenience of retaining his MONITOR statements in the source program without paying the compilation - and resulting object program - price.

A.3.7 DEBUGGING (COMPILER MAINTENANCE)

The JOCIT model includes a wide range of built-in compiler debugging features, mostly in the form of formatted table and file dumps that can be selected individually during maintenance-mode execution of the compiler. The debugging routines themselves occupy symbol table space and are *overwritten* by symbol table entries during production-mode compilation. Thus, the production mode compiler is not enlarged since the maintenance debugging routines are not ordinarily present; however, full debugging capability is available in the compiler by exercising an option.

A.3.8 RELIABILITY

The reported error rate on the production JOCIT J-3 compiler is comparatively low. The errors tend to be distributed throughout the compiler modules, and it is rare for the compiler to fail completely. Not surprisingly, the most vulnerable phase of the compiler is the optimizer since very large programs with complex flow can cause the optimizer to abort. However, in keeping with the diagnostic approach of the design, these failures are (almost without exception) self-detected anomalies. Most optimizer failures relate to unnecessarily complex space management functions which are subject to unpredictable, subtly-compounded errors. The forthcoming JOCIT improvements project, which provides for considerable optimization enhancement, will include simplified restructuring of the optimizer data base and space manager to strengthen this area considerably.

A.4 RETARGETING

In order to achieve retargeting of the present JOCIT model, the following steps are required:

1. Develop library for target machine.
2. Adjust compiler code for target machine sensitivity.
3. Write new direct code processor.

4. Write new code generator.

5. Modify the editor phase to produce object code listing in new target-machine format.

6. Add new object module formatter.

7. Provide for translation of host-machine constant formats to target-machine formats.

These steps are discussed in detail in the following paragraphs.

A.4.1 LIBRARY INSTALLATION

The J-3 library consists mainly of I/O and ENCODE/DECODE routines, string routines, and MONITOR routines. Retargeting requires rewriting these routines for each new target. Except for the MONITOR routines written in SYMPL, these routines are written in assembly code that is not directly transferable to a new target machine. Installation of the library is not a simple task since even the MONITOR routines must be rewritten (unless, of course, a SYMPL compiler exists for the new target machine).

A.4.2 ADJUSTMENT FOR TARGET-MACHINE SENSITIVITY

Many routines within the compiler are affected by various characteristics of the target machine. These are partly parameterized through use of a target-machine descriptor block. However, this parameterization is not yet complete. Typical parameters of interest are:

- Target word size
- Target byte size (bits per byte)
- Target bytes per word
- Maximum and minimum integer values
- Maximum and minimum floating values
- Medium packing access field descriptions
- Addressing units per word

- Character set internal representation
- Target numeric-value representations

The following routines within the compiler presently must be adjusted as described below:

ALOCTR	Object Program Data Allocator - Describe medium packing and type characteristics.
XREF	Cross-Reference Lister - Tailor target-dependent listing.
CCP	Compiler Control Program, i.e., control card scanner - Recognize multiple target option.
JXEC	Compiler Executive or "cradle" - Sequence the compiler as necessary for different target-dependent phases (e.g., the code generator).
COM08T	Target Parameter Data Block - Modify target-machine parameters.
PCON	Constant Posting Routine - Modify as necessary to reflect different internal forms for target.
JINIT	Initialization of Compiler - Post target-specific intrinsic functions, if any (for example, the correct library routine entry point name for the string routines, I/O routines, etc.).
JPF1	Pass 1 Analysis Pragmatic Functions - Convert source form constants to target form.
PF1PR1	Preset Processing Subroutine of Pass 1 Pragmatic Functions - Prepare preset constants in target format.
OPT2A	Pass 2 Optimizer Constant Arithmetic Routine - Modify constant arithmetic to manipulate target form values.

Most of these modifications are individually trivial; however, the number of different routines to be examined and modified makes the composite task moderately complex.

A.4.3 NEW DIRECT CODE PROCESSOR

The direct code processor must be rewritten for each new target-machine assembly language format. This processor is a functionally separate module which must be replaced in the link-edit of the first analysis phase. This necessitates target-machine sensitivity in JXEC to load the proper phase, and requires the maintenance of a unique analysis Phase 1 for each target supported (all but the direct code processor within the phase have the same code for each target machine).

A.4.4 NEW CODE GENERATOR

The major modification for retargeting is the writing of a new code generator. If the level of local optimization and the effective realization of the global optimizations performed by the optimizer are to be sustained, a substantial effort is required.

The basic architecture of the current HIS-6000 code generator may be retained (which considerably reduces the design effort), and much of the machine-independent code (e.g., the triad table builder) need not be rewritten. Still, this must be considered a major task. There will be one code generator for each target machine; JXEC will select the appropriate code generator phase.

A.4.5 EDITOR PHASE MODIFICATION

The editor phase must be modified (there will be a unique editor for each supported target) to generate the proper assembly-like object code listing. The preset-constant processing also must be modified to align values in a manner consistent with the target machine characteristics.

A.4.6 OBJECT MODULE FORMATTER

An object module formatter (actually a part of the editor phase) must be written for each target. The scope of this task is a function of both the complexity of the object module format requirements and the reliability and clarity of the system documentation that describes it. This can range from relatively straightforward to extremely arduous.

A.4.7 TRANSLATION TO TARGET CONSTANT FORMAT

This process has been identified in preceding paragraphs. The problem is to convert from the compiler's internal constant representation to the target machine format. This requirement affects several compiler modules. For example, when the optimizer performs compile time comparisons between character constants, correct inequalities may be computed only when using the target representation, since its collating sequence may not be the same as the host character representation.

A.5 SUMMARY

JOCIT is best described as, first, a competent and serviceable J-3 compiler for the HIS-6000 GCOS machines and, second, a J-3 compiler-building tool. The advantages of JOCIT are:

- Compiler efficiency
- Object code efficiency
- Good diagnostics
- Excellent debugging facilities (both for the user and maintenance team)
- Moderately convenient retargeting
- Use of quick bootstrapping SYMPL compiler tailored to JOCIT needs

The disadvantages are the:

- Size of the compiler
- Changes necessary to make retargeting even less costly
- Changes necessary to make rehosting less costly (a moderately - complex task)
- Reliance on a separate SYMPL compiler that does not take advantage of the JOCIT compiler's own optimization power and requires separate (although rare) maintenance.

APPENDIX B

COMPILER OPTIMIZATION ENHANCEMENTS

This appendix summarizes enhancements to the optimizations presently performed which have been proposed for the JOCIT J-3 compiler. It serves to illustrate the general techniques for extended global optimization.

The optimization scheme now employed is a Level-I LNRA (Linear Nested Region Analyzer) scheme implemented as a two-pass process. Pass 1 (known as OPT1) performs all flow analysis and builds tables defining set and used information for each procedure and loop. Pass 2 (known as OPT2) performs the actual transformations to the code to realize the principal optimizations of common expression elimination, constant arithmetic, code redistribution, and operator strength reduction. The optimizer phases operate on the IL file generated by the analysis phases of the compiler; all transformations are expressed in the IL, and the output is also in the form of an IL file which is subsequently processed by the code generator phase (COGEN). This design permits selectable optimization. If the optimization phases are by-passed as they are when the NOPT option is selected, the IL generated by the front end is passed directly to the code generator. In this mode, only those local optimizations performed by COGEN are effected, and no global optimization is performed at all.

The enhancements discussed in this appendix assume a solution entirely within the framework of the LNRA design. The sum of these improvements will be to raise the level of generated code significantly. It is expected that for any target machine the resultant code will occupy less space and execute in less time. Furthermore, the proposed enhancements do not violate the present target-machine-independent design, and thus the tool concept is not compromised in any way.

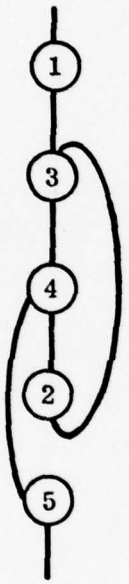
B.1 CODE STRAIGHTENING

In the LNRA method used in JOCIT, program flow is determined in a single forward scan of the program (performed by Optimizer Pass 1, or OPT1). It is the assumption in this approach that a loop is formed whenever a label is reached via a backward branch. Furthermore, loop optimization is suppressed whenever it is observed that a forward branch enters a loop; i.e., redistribution and strength reduction are not attempted on multiple-entry loops (because of the complexity of placing the redistributed and strength-reduced initialization computations in each of the loop's entry blocks). These assumptions are entirely valid on well-structured or "straight" programs. However, they cause the optimizer to miss some cases when the code is not straight. A small program segment demonstrates this point.

The original order of the segment consists of the following five blocks and their interconnecting flow paths:

The optimizer sees 2-3-4 as constituting a loop formed by the backward branch from 4 to 2. However, the forward branch from 1 to 3 defeats the potential loop optimization as described above. Earnest, et al. (10) have proposed an algorithm that may be applied to place the blocks of a program in the straightest possible order.

Its application of the preceding example produces an ordering as follows:



This reordered segment is now a single entry loop (3-4-2), and redistribution and strength reduction algorithms may be applied to Blocks 3 and 4. Block 2, which is conditional, is excluded. The straightening, then, can be seen to have improved the optimization potential.

The code straightening algorithm of Earnest, et al., discovers all program loops in addition to straightening the order. As a result, the code straightener may be used to replace OPT1. Instead of reading the IL, the straightener will

read an extended Global Names List (GNL) file, which will be called the Flow Graph File (FGF). This file will contain each program label, branch, start PROC and end PROC delimiters, PROC call, index switch label list and index switch call. This information is sufficient to identify the basic blocks and interconnecting edges. This representation will then be reordered by the straightener, and the resulting straight order may be represented by an ordered table of "hatchecks" which identify the blocks of the IL to be read in order by OPT2. The FGF will be enhanced further to contain entries for each redefined variable--LHS of assignments, actual value output parameters, and name parameters--such that all redefined variable lists currently produced by OPT1 may be produced by the straightener. Since the FGF is considerably smaller than the IL, it is anticipated that the straightener will be significantly faster than the present OPT1.

B.2 DEAD VARIABLE ANALYSIS

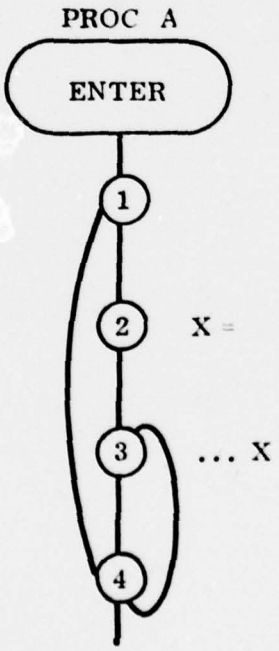
A program variable is said to be dead between its last reference and a subsequent definition. For example, in the program sequence

```
I   . . .
QQ     F(I)$
. . .
I   . . .
```

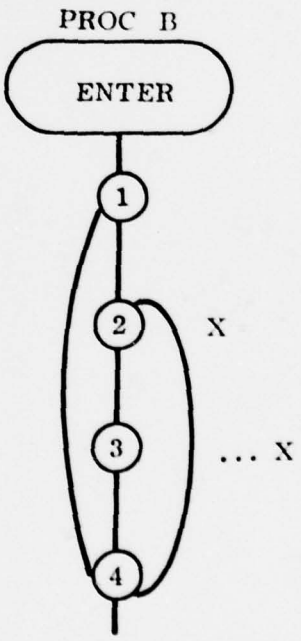
the variable I is dead between the assignment to QQ and redefinition of I in the last line. Recognition of dead variables raises two optimization possibilities which should be examined for cost-effective implementation: (1) store suppression and (2) reuse of dead variable space.

In the above example, the original store into "I" may be suppressed if it is possible to retain "I" in some register between definitions. Following from this, it is seen that retaining "I" in a register means that no storage is required, and the allocated space for "I" may be reused during the program segment where "I" is "dead" by another program variable or compiler-generated temporary.

Dead variable analysis may be conveniently performed by OPT2 after an entire region has been processed. An extension to the definition of a dead variable may include that program segment between a last use and a program exit (or PROC RETURN). However, this may only be for those variables whose first reference in any PROC is a redefinition rather than a reference, i.e., those whose values do not survive from one invocation of the PROC to the next. Since JOVIAL does not permit the programmer to distinguish explicitly between these types of variables, this will be the compiler's task. The analysis procedure is not simple, as the following two procedures demonstrate:



X need not be materialized at PROC exit



X must be materialized at PROC exit

In PROC A there is a path (1-4-3) on which "X" is used before it is set, while in PROC B there is no path to the use of "X" in 3 which does not first redefine "X".

The objectives of dead variable analysis are:

- To suppress unnecessary stores
- To recognize possibilities of allocated space--sharing between programmer variables and compiler-generated temporaries
- To eliminate unnecessary storage allocation for variables held in registers
- To help the code generator retain variables in registers

B.3 LOOP OPTIMIZATIONS

In addition to redistribution and strength reduction optimizations currently performed, loop code can be improved in the following areas:

- Delaying of stores
- Index register dedication of loop control variables (including strength reduction-generated ones)
- Register dedication of redistributed and common values
- Strength reduction test replacement and dead loop control variable elimination
- Strength reduction of addition
- Loop collapse
- Extension of strength reduction to non-FOR loops

These techniques are discussed in the following subsections.

B.3.1 DELAYING STORES

Often with loop code, a variable is repetitively assigned. All but the store immediately preceding loop exit are redundant, and dedicating the variable to a register within the loop and delaying the store into the variable until after loop termination can improve loop performance. For example, in the following program:

```
YY($0$) = 0$
    FOR I = 1,1,99$
        IF XX($I$) GR YY($0$)$
            YY($0$) = XX($I$)$
```

if "YY" is dense-packed, the redundant stores within the loop may be quite costly. The optimizer may recognize the case and transform the program as follows:

```
temp = 0 $
    FOR I = 1,1,99$
        IF XX($I$) GR temp$
            temp = XX($I$)$
                YY($0$) = temp$
```

Thus, in the example, if "temp" is dedicated to a register, both code space and execution time are reduced.

B. 3.2 INDEX REGISTER-DEDICATION OF LOOP VARIABLES

Allocation of loop control variables to index registers eliminates loads and stores within the body of the loop, thus compressing the loop and speeding it up as well. This optimization should be applied both to programmer loop variables and to optimizer-generated loop variables arising from strength reduction. Such dedication may be expressed by means of accenting the

REPL IL operator to indicate that the LHS (the loop variable initialization and increment code, for example) is a candidate for loop dedication. The ENDL operator would signal the code generator to free such loop-dedicated variables.

B.3.3 REGISTER-DEDICATION OF REDISTRIBUTED VALUES

The optimizer moves all redistributed values into the loop entry block. This redistribution is indicated by the VALD operator. The optimizer will mark such redistributed values to make the code generator aware of the motion and to identify the loop from which the values were removed; this will enable the code generator intelligently to select which are the best candidates for register dedication. Even on limited register machines, such as the HIS-6000 series, this can be useful as in the case of a table search, for example:

```
FOR I = 0,1,999$
  IF XX($I$) EQ PATTERN$
     XX($I$) = 0 $
```

In this case, PATTERN may be profitably assigned to an accumulator before the loop (especially helpful if XX is full-word addressable), and the interior of the loop is made smaller and faster. At the current level of optimization, XX(I) is loaded and compared with PATTERN, whereas (assuming XX is full-word addressable) PATTERN may be loaded outside the loop, and only the comparison code is required inside. This same optimization may be performed for values found common and therefore computed outside the loop.

B.4 STRENGTH REDUCTION TEST REPLACEMENT AND DEAD LOOP CONTROL VARIABLE ELIMINATION

During the process of strength reduction, it may be the case that all uses of a loop control variable will have been reduced, such that the loop control variable may be considered dead. In such a case, all references within the body of the loop will have been replaced by generated loop control variables,

and the code to initialize, step, and test the original variable is all that remains. Strength reduction test replacement means to replace the test of the original loop control variable with a derived test on a generated loop control variable. This can be seen from the following simple example

```
FOR I = 0,1,9$
   XX($I*3$) = 0 $
```

which when reduced in strength by the optimizer effectively becomes:

```
temp = 0 $
   FOR I = 0,1,9$
         BEGIN "I"
            XX($temp$) = 0 $
               temp = temp+3$
                  END  "I"
```

If the test against I were replaced by a test against temp, the program could be written:

```
FOR temp = 0,3,27$
   FOR I = 0,1$
      XX($temp$) = 0 $
```

Thus, the use of I is entirely dead, all references are eliminated, and the following simplified and improved program emerges:

```
FOR temp = 0,3,27$
   XX($temp$) = 0 $
```

B.4.1 STRENGTH REDUCTION OF ADDITION

The current strength reduction algorithm includes only the reduction of multiplication and exponentiation. The reduction of addition (which reduces to

another addition) is sensible when it leads to further reduction possibilities. For example, the following program,

```
FOR I = 0,1,99$
  BEGIN "I"
    FOR J = 0,1,99$
      AA($I,J$) = 0 $
        END   "I"
```

in the current JOCIT model reduces only the implicit multiply of J; the subscript expression (I, J) is linearized to ($I+d_1$ *J), where d_1 is the first dimension of AA. Assuming that AA is 100 by 100 (d_1 is then 100) the equivalent code after strength reduction is:

```
FOR I = 0,1,99$
  BEGIN "I"
    t1 = 0 $ "REDUCTION OF 100*J WHICH IS INITIALLY 0"
      FOR J = 0,1,99$
        BEGIN "J"
          AA($I+t1$) = 0 $
            t1 = t1 +100$
              END "J"
                END "I"
```

An improvement to this would result from the reduction of the $I+t_1$ in the inner loop. A straightforward reduction would give:

```
FOR I = 0,1,99$
   BEGIN "I"
      t1 = 0 $
         t2 = I$ "FROM REDUCTION OF I+t WHICH IS
              INITIALLY I"
            FOR J = 0,1,99$
               BEGIN "J"
                  AA($t2$) = 0 $
                     t1 = t1 +100$
                        t2 = t2 +100$
                           END    "J"
                              END  "I"
```

This reduction as shown is actually a degradation of the original program unless the dead loop control analysis is applied along with test replacement. The result is a significant improvement as the following equivalent program shows.

```
FOR I = 0,1,99$
   BEGIN "I"
      FOR t2 = I,100,9900+I$
         AA($t2$) = 0 $
```

NOTE: The expression 9900+I is loop constant over the inner loop, and thus is properly redistributed.

B.4.2 LOOP COLLAPSE

If the preceding example were rewritten with the subscripts reversed,

```
FOR I = 0, 1, 99$
   BEGIN "I"
      FOR J = 0,1,99$
         AA($J,I$) = 0 $          "J,I instead of I,J"
            END "I"
```

the effective reduction looks like the following:

```
t3 = 99$
   FOR t1 = 0,100,9900$
      BEGIN "t1"
         FOR t2 = t1,1,t3$
            AA($t2$) = 0 $
               t3 = t3+100$
                  END  "t1"
```

A close analysis of the above reduction reveals that the inner loop control variable (the generated one, t_2) steps consecutively from 0 to 9999; thus, the inner loop may be collapsed into the outer loop leaving a single loop as follows:

```
FOR t4 = 0,1,9999$
   AA($t4$) = 0$
```

This continuous stepping function may be recognized by observing that the difference between the terminal value of one iteration and the initial value of the next is precisely the step value of the inner loop control. For example, the terminal value of the t_2 on the first iteration is 99 (t_3), and the initial value of the second iteration is 100 (stepped value of t_1); the difference, 1, is the step value for t_2.

A comparison of the various levels of optimization discussed as applied to the simple example discussed above shows the progressive improvements. The examples shown use the HIS-6000 instruction set.

Optimization

Total: 14/Inner loop: 8 (incl. multiply)

```
        STZ     I
L1      STZ     J
L2      LKQ     J
        MPY     100,DL
        ADQ     I
        STZ     AA,QL
        AOS     J
        LDA     J
        CMPA    100,DL
        TMI     L2
        AOS     I
        LDA     I
        CMPA    100,DL
        TMI     L1
```

Optimization - Level 1: Current JOCIT Optimizer

Total: 15/Inner loop: 8 (no multiply)

```
        STZ     I
L1      STZ     J
        STZ     t1              t1 = J*100 = 0
L2      LDQ     I
        ADQ     t1              I+J*100
        STZ     AA,QL
        LDQ     100,DL
        ASQ     t1              t1 = t1+100
        AOS     J
        CMPA    100,DL
        TMI     L2
        AOS     I
        LDA     I
        CMPA    100,DL
        TMI     L1
```

Optimization - Level 2: Reduction of Addition, Test Replacement, Dead Loop Variable Elimination

Total: 15/Inner Loop: 6

```
        STZ     I
L1      LKQ     I
        STQ     t2          t2 = I
        ADQ     9901,DL
        STQ     t3          t3 = 9901+I
L2      LDQ     t2
        STZ     AA,QL       AA($t2$) = 0$
        ADQ     100DL
        STQ     t2          t2 = t2+100
        CMPQ    t3
        TMI     L2
        AOS     I
        LDA     I
        CMPA    100,DL
        TMI     L1
```

Optimization - Level 3: Level 2 + Register Dedication

Total: 11/Inner Loop: 4

```
        LXL1    0,DL        I = 0
L1      EAX2    0,X1        t2 = I
        EAA     9901,X1
        STA     t3          t3 = 9901+I
L2      STZ     AA,X2       AA($t2$) = 0$
        EAX2    100,X2      t2 = t2+100
        CMPX2   t3          test t2 vs. t3
        TMI     L2
        EAX1    1,X1        I = I+1
        CMPX1   10,DU
        TMI     L1
```

Optimization - Level 4: Level 3 + Loop Collapse

Total: 5/Inner Loop: 4

```
        EAX1    0,DU
L1      STZ     AA,X1
        EAX1    1,X1
        CMPX1   9999,DU
        TMI     L1
```

B.4.3 EXTENSION OF STRENGTH REDUCTION TO NON-FOR LOOPS

In the current JOCIT optimizer, strength reduction is applied only to formal (i.e., FOR) loops. Strength reduction may, however, be generalized to include the reduction of functions of any variable satisfying the following conditions:

- The variable is iteratively redefined once only with the scope of some loop; i.e., the variable ("I" for example) is defined by I = I+k or I = I-k
- The variable is nowhere else redefined within the loop
- The redefinition expression (k in the above example) is constant over the same loop in which the variable is iteratively redefined

The recognition of such variables, functions of which are candidates for reduction, will be the responsibility of OPT2 which will employ a double scan program loops identified by OPT1.

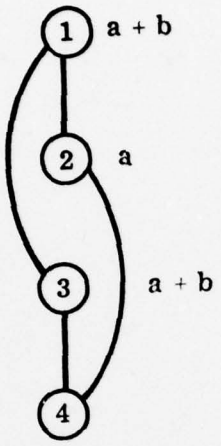
B.5 PARALLEL PATH OPTIMIZATIONS

The IF...THEN...ELSE and CASE constructions in HOLs produce parallel paths in the resulting program flow graph. Certain optimization possibilities arise because of this form of flow which are not addressed in the present JOCIT optimizer. The optimizations considered in the following sections are:

- Improved forward flow analysis
- Load promotion/store delay
- Common name recognition

B.6 IMPROVED FORWARD FLOW ANALYSIS

In the current JOCIT optimizer, all forward branches are treated as conditional insofar as their effect on the state of the search set is concerned. For example, in the following program segment:



the assignment to "a" in Block 2 prevents "a+b" in Block 3 from being recognized as common with that in Block 1 although they compute the same value. The analysis can be improved in OPT2 by restoring search set value numbers at block entrances whose immediate textual predecessor block exits by an unconditional branch.

B.6.1 LOAD PROMOTION/STORE DELAY

In any multipath graph, code space can be saved by preloading common values and by delaying common stores as the following simple case demonstrates:

```
IFEITH BOOL$
   AA($I+J$) = KK*5$
      ORIF 1 $
         AA($I+J$) = KK*5+1$
```

In the current JOCIT compiler, if neither I+J nor KK*5 appear before the IFEITH, the code (shown for the HIS-6000) will be:

```
        LDA     BOOL
        TZE     L1
        LDQ     I
        ADQ     J
        EAX0    0,QL
        LDQ     KK
        MPY     5,DL
        STQ     AA,X0
        TRA     L2
L1      LDQ     I
        ADQ     J
        EAX0    0,QL
        LDQ     KK
        MPY     5,DL
        ADQ     1,DL
        STQ     AA,X0
L2      ...
```

Total 16

Applying the diamond optimizations, the code would be:

```
        LDQ     I
        ADQ     J       PRECOMPUTE I+J
        EAX0    0,QL    SAVE IN X0
        LDQ     KK
        MPY     5,DL    PRECOMPUTE KK*5
                        SAVE IN Q
        LDA     BOOL
        TNZ     L2      TRUE CASE NOW NULL:
L1      ADQ     1,DL    FALSE CASE:
                        COMPUTE KK*5+1
L2      STQ     AA,X0   DELAYED STORE OF
                        AA($I+J$)
```

Total 9 Reduction = 44%

B.7 NAME COMMONALITY

The current JOCIT optimizer employs the value-folding technique to implement common expression recognition. This excludes recognition of certain cases in which expressions are computed on parallel paths which are formally common but which may compute different values. The following program example demonstrates:

```
IFEITH BOOL$
   PP(XX+YY)$
      ORIF 1$ BEGIN
         XX=XX+1$ PP(XX+YY)$
            END
               ...XX+YY...
```

In this example, the expression XX+YY after the IFEITH/ORIF construction is formally common with the XX+YY computed on each path of the diamond; however, since it computes different values--XX is redefined on the ORIF path--the optimizer will not recognize the common case. This problem may be solved within the present value-folding design by the following technique:

- At the terminus of parallel paths, all live names and expressions are permuted for formal name matching. This involves examining the value synonym list and substituting the different name synonyms until the list is exhausted or a match occurs on both paths. In practice, synonym lists for values are quite short, so that this process is not prohibitively slow.
- Common names and expressions so recognized are then assigned to temps on the parallel paths. (e.g., XX+YY is assigned to T1 on both paths in the above example). The same temp is used as a surrogate name for the common expression. This is to give the

expression, on both paths, a common residence which the code generator may subsequently assign to a register in order to achieve the desired optimal effect.

- Common names and expressions thus recognized are now posted to the search set in the usual manner, and the temp is also posted as a synonym for the posted value. Thus, subsequent occurrences of the expression are found common through the conventional folding technique. Dead variable analysis may delete the assignments to the temps on the parallel paths. The optimization is realized by the code generator when it is called to compute the value common subsequent to the parallel network; the temp (which may have been register-dedicated on each path) is chosen as the optimal source of the value.

In the example given above, the value XX+YY is stored in a temp, the same temp, on each path in order for it to be passed as a value parameter in the calls to PP. Thus, references to XX+YY after the diamond may be replaced by references to the temp.

B.8 USE OF COMPOOL BY THE OPTIMIZER

B.8.1 REDEFINED VARIABLE LISTS FOR COMPOOL DEFINED PROCEDURE

Optimization may be enhanced through extension of the COMPOOL concept. The greatest potential payoff is in refining the spoil analysis at COMPOOL-defined procedure calls. Another useful application is in the use of branch frequency information to improve certain loop optimizations.

In the present JOCIT optimizer, calls to local procedures invoke a spoiling process in which only those variables global to the called procedure and also to the point of call are assumed to be redefined by the call. This process

is mechanized in OPT1 which constructs a list of global variables redefined by the procedure. The list also includes other procedures called, so that cascaded effects are accommodated. For calls to externally-defined or COMPOOL-defined procedures, the optimizer makes the assumption that all external and common data are spoiled by the call. This is clearly a safe but overly conservative assumption.

An improvement may be experienced by appending redefined variable list information to the COMPOOL entry for each procedure during COMPOOL assembly. This requires one or both of two other changes to the COMPOOL process. The first requires the programmer to specify in the COMPOOL declaration for the procedure those global variables assigned and those external procedures called. The second approach requires that the COMPOOL source for procedures include the entire procedure body. In this case, OPT1 would be called as part of the COMPOOL assembly, and its constructed RVL would be output with the procedure entry in the COMPOOL.

B.8.2 BRANCH FREQUENCY DATA

As part of the integrated approach to the LCF design, execution measurement data may be subjected to postmortem reduction and the branch frequency data entered into the program COMPOOL. The branch points would be identified by statement number. Thus, subsequent compilations of the program would provide for access to the branch frequency data in that program's COMPOOL by the optimizer for the purposes of replacing branch frequency assumptions by actual operational experience. A limitation inherent in this approach is that source program modifications to the measured program are prohibited between any measured execution and a subsequent recompilation. This guarantees that the statement number data is consistent between the COMPOOL and the subsequent compilation.

B.9 MISCELLANEOUS OPTIMIZATION ENHANCEMENTS

B.9.1 VALUE USAGE CONTEXT

The optimizer will set bits in each VALU entry in the IL to indicate the contexts in which the value is used. These bits may distinguish between computational usage and subscript usage, for example, and will aid the code generator in register allocation for values. For example, a value having only subscript uses may be profitably allocated to an index register. The current JOCIT code generator already makes use of this information, so the setting of these bits yields a low-cost object code efficiency improvement.

B.9.2 UNREFERENCED PROCEDURE DELETION

It frequently happens that as a program ages, certain procedures are no longer referenced, and the programmer fails to remove them. This is an easy case for the compiler to recognize, and the unreferenced procedure need not be compiled.

B.9.3 SINGLE-REFERENCE PROCEDURE OPTIMIZATION

A procedure with only one reference may perhaps be more efficiently compiled as an open routine with actual parameters substituted for formal parameters. This substitution may be performed by OPT2 (or perhaps the straightener) by collecting the actual parameter IL code and substituting it for the corresponding formal parameter IL code, and by eliminating the procedure prologue and epilogue.

B.9.4 REFINED REGION DEFINITION

That segment of the program optimized by the JOCIT compiler in a single unit is known as a region. Currently it is the case that a region break occurs when the optimizer exhausts working storage for the region. When this happens, the

IL is flushed, and working space is recovered for the next region. In the production compiler, working space is sufficient to permit the optimization of several hundred source lines per region.

It is suggested that the region end definition be refined to prevent the termination of a region within a loop that might otherwise be contained. This can be accomplished by the straightener, which will record for each loop entry in the loop list the number of IL entries associated with the loop body. OPT2 may then estimate the working storage requirements - based on the IL count for the loop - whenever a loop top is seen and terminate the region before the loop in the event that working space is insufficient. This will permit the optimization of a loop, or next of loops, within a single region, which the termination of a region in mid-loop prevents (e.g., after such a region end, no strength reduction or redistribution may occur).

B.9.5 REASSOCIATION, DISTRIBUTION, AND CONSTANT COLLECTION

Subscript expressions, linearized by the analysis phase, may be more completely optimized if the rules of reassociation, multiply distribution, and constant reordering are applied. These rules, to be implemented in OPT2, are summarized as follows ("K" stands for any constant, and "&" stands for any associative operator*):

Reassociation

1. The expression $(A \ \& \ K_1) \ \& \ K_2$ is changed to $A \ \& \ K_3$, where $K_3 = K_1 \ \& \ K_2$
2. The expression $(A \ \& \ K) \ \& \ B$ is changed to $(A \ \& \ B) \ \& \ K$
3. The expression $(A \ \& \ K_1) \ \& \ (B \ \& \ K_2)$ is changed to $(A \ \& \ B) \ \& \ K_3$, where $K_3 = K_1 \ \& \ K_2$

*Add and multiply are the only associative operators of interest in this discussion.

Distribution of Multiply

4. The expression $(A + K_1) * K_2$ is changed to $(A * K_2) + K_3$, where $K_3 = K_2 * K_1$

Constant Reordering

5. The expression (K & A) is canonically reordered to (A & K)

The effect of these can be seen on the following two-dimensional array reference XX($1+I,J+1$), where XX is a 10-by-10 array. The linearized subscript is (1+I) + (J+2)*10. The successive application of the above rules to the triads as seen left-to-right is shown:

(1+I) becomes (I+1) by Rule 5.

(J+2)*10 becomes (J*10)+20 by Rule 4.

(I+1)+(J*10)+20) becomes (I+(J*10))+21 by Rule 3.

One of the primary effects of these rules is that any constant offset is floated to the right where it is removed by the code generator and placed in the address field as an address offset.

REFERENCES

1. W. M. Waite and B. K. Haddon, Univerisity of Colorado, "A Preliminary Definition of JANUS", report SEG-75-1 dated September, 1975, prepared for National Science Foundation.

2. W. M. Waite, University of Colorado, "JANUS Memory Mapping: The J1 Abstraction", report SEG-76-1 dated March, 1976, prepared for National Science Foundation.

3. M. C. Newey, P. C. Poole and W. M. Waite, "Abstract Machine Modelling to Produce Portable Software - A Review and Evaluation", Software-Practice and Experience, Vol. 2, pages 107-136 (1972).

4. S. S. Coleman, P. C. Poole and W. M. Waite, "The Mobile Programming System, JANUS", Software-Practice and Experience, Vol. 4, pages 5-23 (1974).

5. Computer Sciences Corporation, "The Genesis System", a report dated May 1967.

6. Computer Sciences Corporation, "Systems Programming Languages (SYMPL)", a report dated January, 1968.

7. Dennis Allison, et al, "Tiny BASIC Status Letters", several articles Dr. Dobbs Journal, Vol I, Nos. 1, 2, 3.

8. R. Whipple and John Arnold, "Tiny BASIC, Extended Version", Dr. Dobbs Journal, Vol I, No. 1 and 2.

9. Tom Pittman, "Tiny BASIC Available for the 6800", Dr. Dobbs Journal, Vol I, No. 2, p. 33.

10. Earnest, et al, "Analysis of Graphs by Ordering of Nodes", JACM, Vol 19, No. 1, January 1972, pp. 23-42.